Online Scheduling of Sequential Task Graphs on Hybrid Platforms

Louis-Claude Canon, Loris Marchal, Bertrand Simon, Frédéric Vivien

Online Scheduling of Sequential Task Graphs on Hybrid Platforms

Louis-Claude Canon, Loris Marchal, Bertrand Simon, Frédéric Vivien

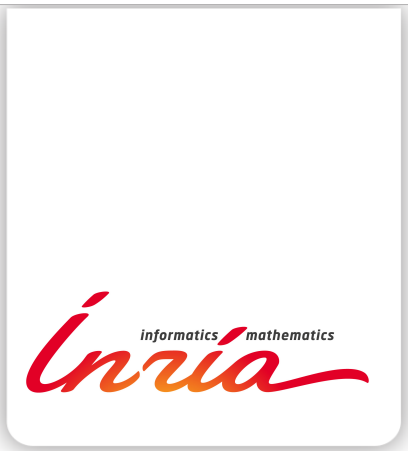
RESEARCH REPORT

N° 9150

Février 2018

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9150--FR+ENG



Online Scheduling of Sequential Task Graphs on Hybrid Platforms

Louis-Claude Canon*, Loris Marchal†, Bertrand Simon‡, Frédéric Vivien§

Project-Team ROMA

Research Report n° 9150 — Février 2018 — 36 pages

Abstract: Modern computing platforms commonly include accelerators. We target the problem of scheduling applications modeled as task graphs on hybrid platforms made of two types of resources, such as CPUs and GPUs. We consider that task graphs are uncovered dynamically, and that the scheduler has information only on the available tasks, i.e., tasks whose predecessors have all been completed. Each task can be processed by either a CPU or a GPU, and the corresponding processing times are known. Our study extends a previous $4\sqrt{m/k}$-competitive online algorithm [3], where m is the number of CPUs and k the number of GPUs ($m \geq k$). We prove that no online algorithm can have a competitive ratio smaller than $\sqrt{m/k}$. We also study how adding flexibility on task processing, such as task migration or spoliation, or increasing the knowledge of the scheduler by providing it with information on the task graph, influences the lower bound. We provide a $(2\sqrt{m/k}+1)$-competitive algorithm as well as a tunable combination of a system-oriented heuristic and a competitive algorithm; this combination performs well in practice and has a competitive ratio in $\Theta(\sqrt{m/k})$. We extend our results to more types of processors. Finally, simulations on different sets of task graphs illustrate how the instance properties impact the performance of the studied algorithms and show that our proposed tunable algorithm performs the best among the online algorithms in almost all cases and has even performance close to an offline algorithm.

Key-words: Scheduling, Task graph, GPU, Hybrid platform, Online

* Louis-Claude Canon is with FEMTO-ST Institute – Université de Bourgogne Franche-Comté
† Loris Marchal is with CNRS, France.
‡ Bertrand Simon is with ENS de Lyon, France.
§ Frédéric Vivien is with Inria, France.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Ordonnancement Online de Graphes de Tâches Séquentielles sur Platformes Hybrides

Résumé : Les plateformes de calcul modernes comportent souvent des accélérateurs. Nous nous intéressons au problème d'ordonnancement d'applications modélisées par des graphes de tâches, sur de telles plateformes composées de deux types de processeurs, par exemple des CPU et des GPU. On considère que les tâches sont dévoilées dynamiquement, et que l'ordonnanceur ne connaît que les tâches disponibles, i.e., les tâches dont les prédecesseurs ont tous été exécutés. Chaque tâche peut être traitée soit par un CPU soit par un GPU, et les temps de calculs correspondants sont connus. Notre étude étend un précédent algorithme *online* $4\sqrt{m/k}$-competitif, où m est le nombre de CPU et k le nombre de GPU ($m \geq k$). Nous prouvons qu'aucun algorithme *online* ne peut avoir un facteur de compétitivité plus petit que $\sqrt{m/k}$. Nous étudions également comment cette borne inférieure est influencée par l'ajout de flexibilité sur le traitement des tâches (migration ou spoliation) ou par une meilleure connaissance du graphe par l'ordonnanceur. Nous fournissons un algorithme $(2\sqrt{m/k}+1)$-compétitif ainsi qu'une combinaison paramètrable avec un algorithme efficace en pratique qui permet d'obtenir un facteur de compétitivité en $\Theta(\sqrt{m/k})$. Nous étendons nos résultats pour plus de deux types de processeurs. Enfin, des simulations sur plusieurs ensembles de graphes de tâches illustrent les performances des algorithmes étudiés.

Mots-clés : Ordonnancement, Graphe de tâches, GPU, plateforme hybride, online

Contents

1 Introduction

Modern computing platforms increasingly use specialized hardware accelerators, such as GPUs or Xeon Phis: 102 of the supercomputers in the TOP500 list include such accelerators, while several of them include several accelerator types [21]. The increasing complexity of such computing platforms makes it hard to predict the exact execution time of computational tasks or of data movement. Thus, dynamic runtime schedulers are often preferred to static ones, as they are able to adapt to variable running times and to cope with inaccurate predictions. Indeed, with the widespread heterogeneity of computing platforms, many scientific applications now rely on runtime schedulers such as OmpSs [19], XKaapi [8], or StarPU [5]. Most of these frameworks model an application as a Directed Acyclic Graph (DAG) of tasks, where nodes represent tasks and edges represent dependences between tasks. While task graphs have been widely studied in the theoretical scheduling literature [12], most of the existing studies concentrate on static scheduling in the offline context: both the graph and the running times of the tasks are known beforehand.

We believe that there is a crucial need for online schedulers, that is, of scheduling algorithms that rely neither on the structure of the graph nor on the knowledge of tasks' running times. First, not all graphs are fully available at the

beginning of the computation: sometimes the graph itself depends on the data being processed, different inputs may result in different task graphs. This is especially the case when the behavior of an iterative application depends on the accuracy of the output. Second, in most existing runtimes, even if the graph does not depend on the input data, it is not fully submitted at the beginning of the computation; instead, tasks are dynamically uncovered during the computation. Third, even if part of the graph is available, schedulers usually avoid traversing large parts of the graph each time they take a decision in order to strongly limit the time needed to take decisions. Finally, tasks' processing times are not always known beforehand, and the occasionally available predictions may not be very accurate, as two successive executions of the same task may result in slightly different timings.

There has recently been an effort of the scheduling community to fill the gap between the assumptions used in theoretical studies and those underlying schedulers for runtime systems (see details in Section 2). Schedulers for independent tasks on hybrid platforms have first been proposed [6, 9, 17]. Recently, an online scheduler for independent tasks on hybrid platforms [15] has been adapted for task graphs [3].

In the present paper, we concentrate on the online scheduling of task graphs on a hybrid platform composed of 2 types of processors, that we call CPU and GPU for convenience. There are m CPUs and k GPUs, where $m \geq k \geq 1$. Note that we do not make any assumptions on the CPUs and GPUs, so that these results may be symmetrically applied to the converse case with more GPUs. The objective is to schedule a DAG G of tasks, so as to minimize the total completion time, or makespan. Each task can be assigned either to a single CPU or to a single GPU. We adopt the notations of [3]: the processing time of task T_i on a CPU is noted by $\overline{p_i}$ and on a GPU by $\underline{p_i}$.

We consider the following online problem. At the beginning, the algorithm is aware of all the input tasks of the graph, and can schedule each one on either a CPU or on a GPU. A task is released and becomes available to the scheduler only when all its predecessors are terminated. At any given point in the computation, the scheduler is totally unaware of tasks which have not yet been released, but it knows the processing times $\overline{p_i}$ and $\underline{p_i}$ of all available tasks. No delay is necessary between the release of a task and the start of its processing, hence we do not take into account the time needed for moving data.

The closer related work considering the very same problem is [3] which provides a $4\sqrt{m/k}$-competitive algorithm for this problem. The number $\sqrt{m/k}$ will be used throughout the paper as it appears to be deeply connected to this problem. We will therefore use the notation $\tau = \sqrt{m/k}$. The present report is organized as follows:

- In Section 3, we prove that the competitive ratio of any online algorithm is lower-bounded by $\tau = \sqrt{m/k}$. We study how the knowledge of the task graph and the flexibility of the tasks may influence the lower bound; we

especially prove that knowing the bottom level of any task or having preemptive tasks does not help much, whereas the knowledge of the number of descendants allows to reduce the lower bound to $\frac{1}{2}\sqrt{\tau}$.
- In Section 4.1, we propose a $(2\tau+1)$-competitive algorithm, by refining both the algorithm and the analysis of [3].
- In Section 4.2, we propose a simple heuristic, based on the system-oriented heuristic EFT, which is both a competitive algorithm and performs well in practice.
- In Section 5, we study the complexity of the online problem with an oracle providing either the optimal allocation or the optimal schedule. We show that knowing the allocation allows to design 3-competitive algorithm, whereas no τ-competitive algorithm can decide online the allocation even if the optimal schedule is provided by an oracle.
- In Section 6 we study the generalized problem with more than two types of processors. We extend both the lower bounds and the online algorithm of Section 4.1.
- In Section 7, we study through simulations the behavior of several online algorithms on different datasets, composed either of actual or synthetic task graphs.

2 Related Work

We briefly position our contributions in comparison to the existing work, starting with the offline case when the whole scheduling problem (both task dependences and running times) is known beforehand.

Offline algorithms. Several schedulers for independent tasks on hybrid platforms have been proposed. Bleuse et al. [17] designed a polynomial but expensive $\left(\frac{4}{3}+\frac{1}{3k}\right)$-approximation. Low complexity algorithms, which are closer to our work, have been studied in [6, 9] and achieve approximation ratios respectively equal to 2 and $2+\sqrt{2}$. For tasks with precedence constraints, Kedad-Sidhoum et al. [16] provided a tight 6-approximation based on linear programming.

Online algorithms. When tasks with precedences are released over time, Graham's List Scheduling algorithm [14] is 2-competitive on homogeneous processors (note that this is also the best offline approximation for this problem). On our model with two sets of processors, Imreh [15] and Chen et al. [11] proposed an algorithm to schedule independent tasks with a competitive ratio smaller than 4. Based on this work, Amaris et al. [3] exhibited an online algorithm for precedence constraints, achieving a competitive ratio of $4\sqrt{m/k}$.

Runtime strategies. Actual runtime schedulers usually rely on low-complexity scheduling policies to limit the time needed to allocate tasks. For instance, StarPU [5] builds a performance model of tasks that allows it to predict their

processing times. When a new task is submitted, it is allocated to the resource that will complete it the soonest (when using the **dm** policy, previously called **heft-tm** in [4]), which corresponds to the classical Earliest Finish Time (EFT) scheduling policy [18]. Other strategies have been proposed that take into account communication times, or precomputed task priorities, depending on the descendants of each task. We include similar information in the design of the lower bounds on competitive ratios (Section 3).

3 Lower bound on online algorithms competitiveness

In this section, we provide a lower bound on the competitive ratio of any online algorithm: no online algorithm has a competitive ratio smaller than $\tau = \sqrt{m/k}$ (Theorem 1). We also study how adding flexibility to task processing or giving some knowledge of the graph to the scheduler impacts this lower bound.

Intuitively, the main difficulty for this problem arises from choosing on which type of resource (CPU or GPU) a given task should be processed, and not to come up with the final schedule. This is indeed proven in Theorem 6, Section 5: if the allocation of the tasks is fixed, any online list scheduling algorithm is $\left(3-\frac{1}{m}\right)$-competitive, which is optimal.

The proof of Theorem 1 heavily relies on the fact that an online algorithm has no information on the successors of each task. In practice, it is sometimes possible to get some information on the task graph, for example by precomputing some information offline before submitting the tasks. For instance, offline schedulers usually ranks available tasks with priorities based on the dependences. On homogeneous platforms, the *bottom-level* of a task is commonly used, and is defined as the maximum length of a path from this task to an exit node, where nodes of the graphs are weighted with the processing time of the corresponding tasks. In the heterogeneous case, the priority scheme used in the standard HEFT algorithm [22] is to set the weight of each node as the average processing time of the corresponding task on all resources.

Knowing the bottom-level does not change the lower-bound of Theorem 1, see Theorem 2. The only benefit is a diminution by a factor 2 if there is exactly one GPU. An interesting component of this proof is that all the tasks are equivalent (same CPU and GPU computing times) so other heterogeneous variants of the bottom level are also captured.

When the online scheduler is given the knowledge of the number of descendants of each submitted task in addition to their bottom-level, the lower bound of Theorem 1 is reduced to $\frac{1}{2}\sqrt{\tau}$ when m/k is large enough (see Theorem 3), so no constant-factor competitive algorithm exists. Note that all the tasks are equivalent in this proof; so it also captures, for instance, the knowledge of the CPU and GPU computing times of all the descendants; only the pattern of precedence relations remains unknown. Note that, however, no algorithm has been proposed that reaches this bound.

Another interesting question is whether adding flexibility on how tasks are processed changes this bound. Allowing task spoliation (where tasks can be canceled and restarted on another resource, as done in [6]) does not change any lower bound. Allowing task migration (where tasks can be interrupted and resumed on another resource) divides the lower bounds obtained by a factor 2.

Table 1 summarizes the results for all combination of knowledge given to the scheduler and flexibility on the task processing. The best competitive algorithm for every setting is smaller than $2\tau + 1$, and is achieved by QA. This algorithm does not use all the knowledge or flexibility as it does not practice spoliation or migration, does not use any information on the bottom level or the descendants, and schedule tasks one by one, without looking at other available tasks.

Flexibility	Knowledge	Lower bound	Proof	Note
None or Spoliation	None	τ	Th. 1	
	Bottom Level	τ	Th. 2	$\frac{1}{2}\tau$ if $k=1$
	BL + descendants	$\frac{1}{2}\left\lfloor\sqrt{2\tau^*}\right\rfloor$	Th. 3	
Migration	None	$\frac{1}{2}\tau$	Th. 1	
	BL	$\frac{1}{2}\tau$	Th. 2	$\frac{1}{4}\tau$ if $k=1$
	BL + descendants	$\frac{1}{4}\left\lfloor\sqrt{2\tau^*}\right\rfloor$	Th. 3	

Table 1: Summary of the results obtained for various versions of online models. τ^* represents the largest triangular number such that $\tau^* \leq \tau$. Note that $\left\lfloor\sqrt{2\tau^*}\right\rfloor \geq \sqrt{\tau}$ for large values of τ.

First, we consider algorithms that are not aware of the bottom level of the tasks.

Theorem 1. *No online algorithm has a competitive ratio smaller than τ, even when spoliation is authorized. If preemption with migration is authorized, no online algorithm has a competitive ratio smaller than $\frac{\tau}{2}$.*

Proof. Consider an online algorithm $\mathscr{A}$, making use of spoliations. We assume for the moment that τ is an integer. We consider an integer n as large as we want. A large n will lead to a large graph and a competitive ratio closer to τ. We will use an adversary proof, by building a graph composed of nm tasks denoted by T_i^j, with $1 \leq j \leq n\tau$ and $1 \leq i \leq k\tau$. The CPU processing time of each task equals τ and the GPU processing time equals 1.

The procedure can be cut into $n\tau$ phases. During the jth phase, tasks T_i^j for i from 1 to $k\tau$ are independent and available. The adversary selects the task that $\mathscr{A}$ completes the latest, breaking ties arbitrarily. Let T_*^j be this task. The $k\tau$ tasks of the next phase are then made successors of T_*^j. See Figure 1 for an illustration of a built graph.

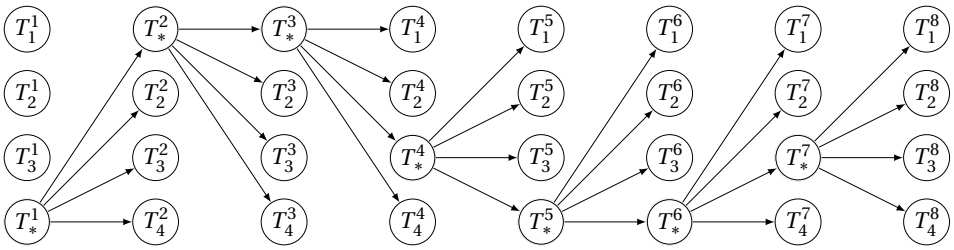


Figure 1: Example of built graph with $\tau = 2$, $k = 2$, $n = 4$.

We define a schedule $\mathscr{S}$ of the built graph. We define a *bucket* as a set of processors, a starting time and a duration time. We use buckets to reserve some processors for an amount of time, and schedule a set of tasks in a given bucket. We consider $n+1$ buckets, as illustrated in Figure 2. Buckets B_i for i from 1 to n each concerns all m CPUs, lasts a time τ, and starts at time $i\tau$. Note that m tasks fit into each bucket. The last bucket, B concerns one GPU, starts at time 0 and lasts a time $n\tau$.

$\mathscr{S}$ schedules the $n\tau$ tasks T_*^j successively on a single GPU, which fit into bucket B. In parallel, $\mathscr{S}$ schedules the remaining tasks on CPU. More precisely, schedule in bucket B_ℓ the tasks T_i^j such that $(\ell-1)\tau < j \leq \ell\tau$, except the tasks T_*^j. They all fit into the bucket as there are less than $\tau * k\tau \leq m$ such tasks. Moreover, task $T_*^{\ell\tau}$ is completed at time $\ell\tau$, so every task T_i^j with $j \leq \ell\tau$ can be initiated after time $\ell\tau$, so can be scheduled into bucket B_i. Therefore, $\mathscr{S}$ achieves a makespan equal to $(n+1)\tau$.

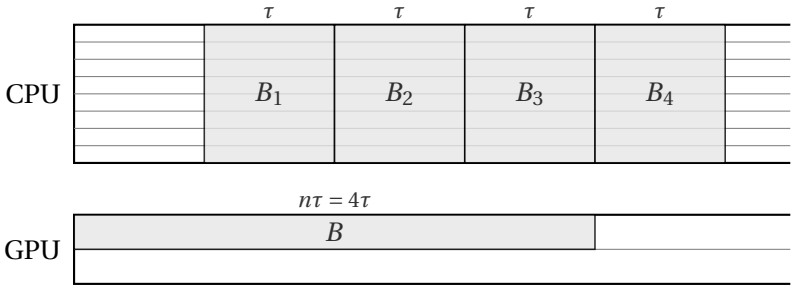


Figure 2: Buckets used by $\mathscr{S}$ with $n = 4$.

Now, we consider algorithm $\mathscr{A}$, and we show that the makespan obtained is at least $n\tau^2$. At each phase, the adversary reveals the next phase only when all the tasks of the current phase are completed. If one task of the phase is scheduled on CPU, it takes a time τ. Otherwise, all $k\tau$ tasks are scheduled on GPU, and the last one completes at time at least $k\tau/k = \tau$. Therefore, $\mathscr{A}$ completes each phase in time at least τ. Note that making use of spoliation is not useful in this case. As there are $n\tau$ phases, the whole graph cannot be scheduled in time smaller than $n\tau^2$. The competitive ratio of $\mathscr{A}$ is then at least:

$$\frac{n\tau^2}{(n+1)\tau} \quad \underset{n\to\infty}{\longrightarrow} \quad \tau.$$

Now, consider an algorithm $\mathscr{A}$' which makes use of preemption with migration. The adversary strategy and the schedule $\mathscr{S}$ is unchanged. We first prove by contradiction that $\mathscr{A}$' cannot terminate a phase in a makespan smaller than $\tau/2$. Assume that one phase is terminated in time $\tau/2$. We consider the fraction of each task performed on a CPU. All tasks have a processing time of τ on CPU, so for each task, this fraction cannot be larger than one half. Therefore, at least half of each task is executed on a GPU, which takes a time $1/2$ for each task, so it takes $k\tau/2$ units of GPU computing time. As we assumed that the phase is terminated in time $\tau/2$, there is no more than $k\tau/2$ work units available on the k GPUs, which thus cannot process more than one half of each task. Therefore, at least half of each task is processed on CPUs, from the very beginning to the very end of the phase. This requires to execute each task simultaneously on a CPU and on a GPU, which is not possible even with migration. Therefore, $\mathscr{A}$' cannot terminate one phase in time $\tau/2$ (and a fortiori in a shorter time). Thus, $\mathscr{A}$' requires a time larger than $n\tau^2/2$ to complete all $n\tau$ phases. The competitive ratio of $\mathscr{A}$' is then at least:

$$\frac{\frac{1}{2}n\tau^2}{(n+1)\tau} \quad \underset{n\to\infty}{\longrightarrow} \quad \frac{\tau}{2}.$$

In a last step, we now relax the constraint that τ is an integer. Let q be an integer as large as we want, and $r = \lfloor q(\tau - \lfloor\tau\rfloor)\rfloor$, so that $r/q \leq \tau - \lfloor\tau\rfloor \leq (r+1)/q$. A large q will lead to a greater precision. We adapt the graph in the following way: there are now $n(\lfloor\tau\rfloor + r)$ phases each containing $k\lfloor\tau\rfloor + 1$ tasks. For each phase j such that $(j \mod n) \leq \lfloor\tau\rfloor$, the tasks have a CPU computing time τ and a GPU computing time 1. In the remaining nr phases, tasks have a CPU computing time equal to τ/q and a GPU computing time equal to $1/q$. Intuitively, we split the phases corresponding to the fractional part of τ into multiple phases of smaller tasks.

We now adapt the schedule $\mathscr{S}$ which still fits inside the buckets (as previously defined). The tasks T_*^j all fit inside bucket B. Indeed, there are composed of $n\lfloor\tau\rfloor$ tasks of GPU computing time 1 and nr tasks of GPU computing time $1/q$; the time needed to process them sequentially is then equal to $n(\lfloor\tau\rfloor + r/q) \leq n\tau$. For bucket B_1, we execute the tasks T_i^j for $j = 1,\ldots,\lfloor\tau\rfloor + r$ and $i = 1,\ldots,k\lfloor\tau\rfloor + 1$, except tasks T_*^j. The corresponding tasks T_*^j are completed before the start of bucket B_1. Inside bucket B_1, we execute $k\lfloor\tau\rfloor^2$ tasks of CPU computing time τ and $kr\lfloor\tau\rfloor$ tasks of CPU computing time τ/q. The number of processors needed is then:

$$\begin{aligned} k\lfloor\tau\rfloor^2 + \left\lceil k\frac{r}{q}\lfloor\tau\rfloor\right\rceil &\leq k\lfloor\tau\rfloor^2 + \lceil k(\tau - \lfloor\tau\rfloor)\lfloor\tau\rfloor\rceil \\ &\leq k\lfloor\tau\rfloor^2 - k\lfloor\tau\rfloor^2 + \lceil k\tau\lfloor\tau\rfloor\rceil \leq \lceil k\tau^2\rceil = m. \end{aligned}$$

Therefore, the first $\lfloor\tau\rfloor + r$ phases fit into bucket B_1. The same reasoning applies to the following buckets, so $\mathscr{S}$ achieves a makespan equal to $(n+1)\tau$.

Concerning the algorithm $\mathscr{A}$ (assuming it does not make use of preemption with migration), the phases where tasks have a GPU computing time equal to 1 still need a time τ to be completed: if one task is scheduled on CPU, it takes a time τ; if all $k\lfloor\tau\rfloor + 1$ tasks are scheduled on GPUs, this takes at least a time $\lfloor\tau\rfloor + 1 \geq \tau$. Similarly, the other phases need a time τ/q to be completed. The total makespan is then at least $n\tau\left(\lfloor\tau\rfloor + \frac{r}{q}\right) \geq n\tau\left(\tau - \frac{1}{q}\right)$. When q and n tend to infinity, the competitive ratio tends to τ. Note that if $\mathscr{A}$ makes use of preemption with migration, this ratio tends to $\tau/2$, which terminates the proof. □

The proof of Theorem 1 heavily relies on the fact that the online algorithm has zero information on the successors of each task. In practice, it is sometimes possible to get some information, such as the bottom level of each task. On heterogeneous platforms, several adaptations exist to the bottom level, as discussed at the beginning of this section. We prove that using such information does not improve the bound.

Theorem 2. *If $k \geq 2$, no online algorithm has a competitive ratio smaller than τ, even when spoliation is authorized, and the bottom level of each task is known. If preemption with migration is authorized, no online algorithm has a competitive ratio smaller than $\frac{\tau}{2}$.*

If $k = 1$, then we obtain the same bounds divided by a factor 2.

Proof. The proof relies on the construction used to prove Theorem 1. We assume for the moment that $k \geq 2$. For simplification, we rely on the construction for an integer τ but the modification easily extends to a decimal τ.

We add $n\tau$ tasks U^j to the built graph, with $1 \leq j \leq n\tau$, where there is a dependence from U^j to U^{j+1} for each j. Each task has a CPU computing time equal to τ and a GPU computing time equal to 1, as tasks T_i^j. For each task T_i^j, we add a dependence from T_i^j to U^j. See Figure 3 for an illustration of the graph.

The longest path starting from any task T_i^j to an endpoint of the built graph then has a length equal to $n\tau - j + 2$: it is composed for instance of task T_i^j and tasks U^j to $U^{n\tau}$. Note that tasks T_*^j have multiple paths of length $n\tau - j + 2$, see Figure 3.

Therefore, for any j, the $k\tau$ tasks T_i^j have the same bottom level. So when $\mathscr{A}$ terminates task T_i^j, the adversary can choose whether T_*^j is equal to T_i^j or not, while respecting the bottom level furnished to $\mathscr{A}$. Then, the lower bound on the makespan reached by $\mathscr{A}$ (and $\mathscr{A}$' if preemption with migration is allowed) still holds.

It remains to define the schedule $\mathscr{S}$ with the added tasks, and to show that its makespan is at most $(n+2)\tau$. Recall that we assumed $k \geq 2$. We add another bucket B^U concerning a different GPU than B (as $k \geq 2$), starting at time 2τ and

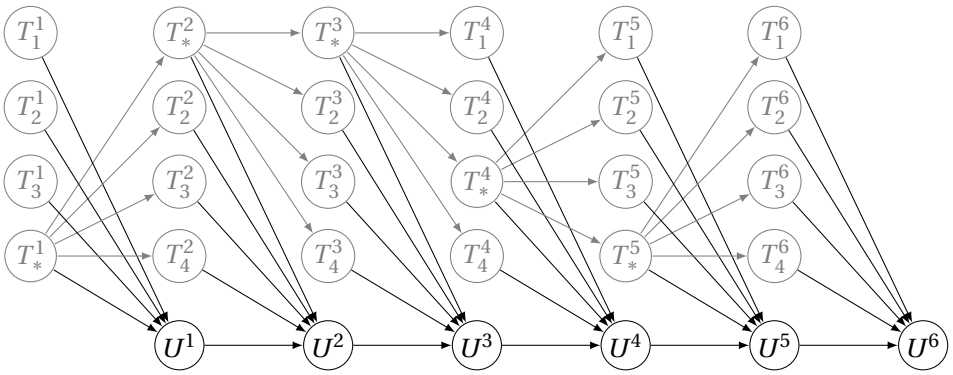


Figure 3: Example of built graph with $\tau = 2$, $k = 2$, $n = 3$. In gray, the tasks and dependences existing in the previous proof.

lasting $n\tau$ units of time, see Figure 4. Task U^j is scheduled in B^U at time $2\tau + j$. Note that for any ℓ, tasks $U^{(\ell-1)\tau}$ to $U^{\ell\tau}$ are executed after bucket B_ℓ, which contains tasks T_i^j for $(\ell-1)\tau < j \leq \ell\tau$. Therefore, every task T_i^j is terminated before task U^j is scheduled, so no precedence constraints are violated.

The lower bounds proved in the Theorem 1 are then unchanged.

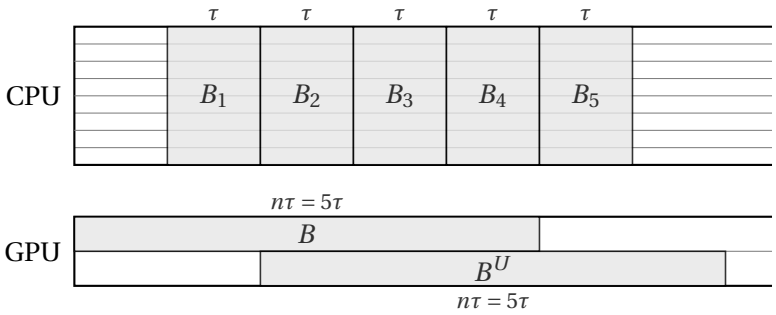


Figure 4: Buckets used by $\mathscr{S}$ with $n = 5$.

If $k = 1$, we define the bucket B^U on the unique GPU, starting at time $n\tau$ and terminating at time $2n\tau$. The makespan obtained by $\mathscr{S}$ is then twice longer, so the lower bounds obtained are twice smaller.

□

In the proof of Theorem 2, we used the fact that the bottom level is no longer useful to differentiate tasks T_*^j from other tasks T_i^j. A metric that could still be used is the total weight of the descendants of a task. We however prove in Theorem 3 that this knowledge only improves the lower bound by a factor two. As discussed at the beginning of this section, all the tasks used in the following proof are identical, so any the weight can capture several functions such as the number of descendants, the average computing time...

Theorem 3. *No online algorithm has a competitive ratio smaller than* $\frac{1}{2}\lfloor\sqrt{2\tau^*}\rfloor$*, even when spoliation is authorized, and both the bottom level and the total weight of the descendants of each task is known. If preemption with migration is authorized, no online algorithm has a competitive ratio smaller than* $\frac{1}{4}\lfloor\sqrt{2\tau^*}\rfloor$*.*

In these bounds, τ^* *is the largest triangular integer not larger than* τ*. Recall that* τ *is a triangular integer if we have* $\tau = 1+2+\ldots+\lfloor\sqrt{2\tau}\rfloor$*.*

Proof. The proof relies on the construction used to prove Theorem 2, but using less phases and adding several tasks. We first assume that τ is a triangular integer larger than 1, which means that there exists an integer $q > 1$ such that $\sum_{i=1}^{q} i = \tau$. The exact value of q is $\frac{1}{2}\sqrt{1+8\tau} - \frac{1}{2} = \lfloor\sqrt{2\tau}\rfloor$. The graph built in this proof contains $q+1$ phases.

We add m tasks V_i^j to the built graph, with $1 \leq j \leq q$ and for each j, with $1 \leq i \leq (q+1-j)k\tau$. By definition of q, this indeed sums to $k\tau^2 = m$ additional tasks. All these tasks have a CPU computing time equal to τ and a GPU computing time equal to 1, as tasks T_i^j.

We now build the graph so that for each j, the $k\tau$ tasks T_i^j have the same number of successors. For each $j \leq q$, we add dependences from every task T_i^j except T_*^j to every task $V_i^{j'}$ such that $j' \geq j$. See Figure 5 for an illustration of the graph, where all tasks T_*^j have been set to $T_{k\tau}^j$ for simplification, and where tasks sharing the same successors and predecessors have been agglomerated. In this example, we have $q = 3$, $\tau = 6$ and $m = 36$. For instance T_1^3 have 6 new successors, tasks $V_{1\ldots6}^3$, T_1^2 has 18 new successors and T_1^1 has $m = 36$ new successors.

Let j be fixed. In this graph, the descendants of task T_*^j are tasks $U^{j'}$ for $j' \geq j$, tasks $T_i^{j'}$ for $j' \geq j$ and tasks $V_i^{j'}$ for $j' > j$. The descendants of any task T_i^j except T_*^j are tasks $U^{j'}$ for $j' \geq j$ and tasks $V_i^{j'}$ for $j' \geq j$. The difference is that task T_*^j has the $(q+1-j)k\tau$ tasks $T_i^{j'}$ as successors but not the $(q+1-j)k\tau$ tasks V_i^j. Therefore, at j fixed, the number of successors is the same for each task T_i^j.

So when $\mathscr{A}$ terminates a task T_i^j, the adversary can choose whether T_*^j is equal to T_i^j or not, while respecting the bottom level and number of descendants furnished to $\mathscr{A}$. Then, the lower bound on the makespan reached by $\mathscr{A}$ (and $\mathscr{A}$' if preemption with migration is allowed) on each phase still holds. Therefore, $\mathscr{A}$ leads to a makespan of at least $(q+1)\tau$ and $\mathscr{A}'$ to a makespan of at least $\frac{1}{2}(q+1)\tau$.

It remains to define the schedule $\mathscr{S}$ with the added tasks, and to show that its makespan is at most $2\tau + q + 1$. The schedule $\mathscr{S}$ is similar to the one of the previous proof, except that tasks V_i^j are executed on CPU after tasks T_i^j. As there are m such tasks, this takes a time τ. To summarize, tasks T_*^j are executed on GPU in time q, then the remaining tasks T_i^j are executed on CPU in time τ, then, in parallel, tasks U^j are executed on GPU in time $q+1$ and tasks V_i^j are executed

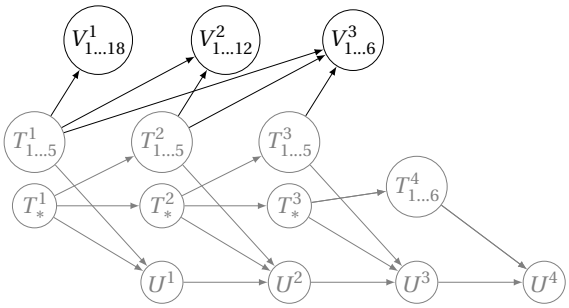


Figure 5: Example of built graph with $\tau = 6$, $q = 3$, $k = 1$, $m = 36$. In gray, the tasks and dependences existing in the previous proof.

on CPU in time τ, see Figure 6. The makespan obtained is then equal to $\tau + q + \max\{\tau, q+1\} = 2\tau + q$. The last equality is valid as for $\tau \geq 3$, we have $q \leq \tau - 1$.

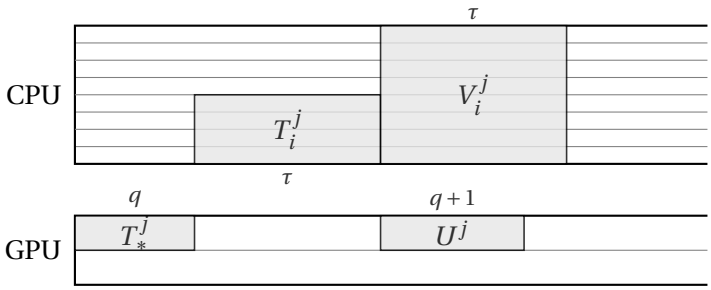


Figure 6: Shape of the schedule $\mathscr{S}$.

Recalling that $\tau = q(q+1)/2$, the competitive ratio of $\mathscr{A}$ is then at least:

$$\frac{(q+1)\tau}{2\tau+q} = \frac{1}{2}\frac{q(q+1)^2}{q(q+1)+q} > \frac{q}{2}\frac{(q+1)^2}{(q+1)^2} = \frac{q}{2} > \frac{1}{2}\left\lfloor\sqrt{2\tau}\right\rfloor.$$

Similarly, the competitive ratio of $\mathscr{A}$' is at least $\frac{1}{4}\left\lfloor\sqrt{2\tau}\right\rfloor$.

If τ is not a triangular integer, or even not an integer, the same proof applies where q is the maximal integer such that $\sum_{i=1}^{q} i \leq \tau$. The exact value of the computing ratio obtained is then:

$$\frac{1}{2}\left\lfloor\sqrt{2\tau+\frac{1}{4}}-\frac{1}{2}\right\rfloor = \Theta\left(\sqrt{\tau}\right).$$

□

4 Competitive algorithms

4.1 The Quick Allocation (QA) algorithm

Amaris et al. [3] designed an online algorithm named ER-LS which can be described as follows:

1. Take any available task T_i.
   (a) If T_i can be terminated on GPU in the current schedule before time $\overline{p_i}$, assign it to GPU.
   (b) If $\overline{p_i}/\underline{p_i} \leq \tau = \sqrt{m/k}$, then assign T_i to CPU, otherwise assign it to GPU.
2. Schedule T_i as soon as possible on the assigned type of processor
3. If there are remaining tasks, return to Step 1.

This algorithm is proved to be 4τ-competitive. We propose here a simplified version of this algorithm, for which we prove a better competitive ratio. The improvement comes both from the simplification and a tighter analysis. We define the algorithm QA, which stands for Quick Allocation. Its allocation phase consists exclusively in the second part of algorithm ER-LS:

- Take any available task T_i. If $\overline{p_i}/\underline{p_i} \leq \tau$, then assign T_i to the CPU side, otherwise assign it to the GPU side.

Note that this allocation phase does not take into account any precedence relation or current schedule. Once this task is allocated to the CPU or GPU side, it is scheduled on the processor of this side which has completed its tasks the soonest.

One could wonder why the ratio τ is the best choice in the allocation phase. Intuitively, there are more CPUs than GPUs, so if $\overline{p_i}/\underline{p_i} < 1$, task T_i is executed faster on CPU, which is a lesser rare resource, therefore task T_i should be allocated to CPU. On the contrary, if $\overline{p_i}/\underline{p_i} > m/k$, then not only task T_i is executed faster on GPU, but if there are many independent tasks T_i to compute, they will be executed faster all on GPUs than all on CPUs. Therefore we can allocate T_i to GPU without wasting a rare resource. When this ratio is between 1 and m/k, the loss is minimized when switching resource at the geometric mean of 1 and m/k, which is equal to τ.

We now show that QA is $\left(2\tau+1-\frac{1}{k\tau}\right)$ – competitive and that this ratio is almost tight, as we provide an example on which QA leads to a makespan $\left(2\tau+1-\frac{1}{k}\right)$ times larger than the optimal solution.

Theorem 4. QA *is* $\left(2\tau+1-\frac{1}{k\tau}\right)$ *– competitive.*

Proof. We consider a graph, an online instance of this graph, and the schedule $\mathscr{S}$ computed by QA, of makespan C_{max}. We also consider an optimal schedule of this graph (later referred to by *the* optimal solution), and we let OPT be its makespan.

Let W_c (resp. W_g) be the total load on the CPUs (resp. GPUs). Let cp be a critical path the task graph given the allocation of $\mathscr{S}$, and CP be the sum of the processing times of the tasks of cp in $\mathscr{S}$.

The objective is to prove that:

$$C_{max} \leq \left(2\tau + 1 - \frac{1}{k\tau}\right)\text{OPT}.$$

We first use Lemma 1 to bound C_{max} using the processor loads (W_c and W_g) and the length of the critical path (CP). Then, we bound the expression obtained in function of OPT to prove the result.

Lemma 1.

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1 - \frac{1}{m}\right) CP.$$

Proof. We consider the path p defined as being the longest path (in terms of execution time in $\mathscr{S}$) that contains a task that terminates at time C_{max} in $\mathscr{S}$. By definition, the length of p is at most CP. In order to simplify the reasoning, we assume that there is a task T_0 of null processing time that is the predecessor of every task in the graph, and that this task belongs to p.

Consider a moment t when no task of p is being executed in $\mathscr{S}$. Let T_ℓ be the last task of p to be executed before t and T_n be the next task of p to be executed after t in $\mathscr{S}$. Note that both tasks always exist because T_0 is executed at the start of the graph and a task of p is executed at the end of the schedule $\mathscr{S}$. Suppose first that T_n is executed on CPU. As T_n is not scheduled immediately after T_ℓ, and the schedule has been obtained by a list algorithm, no CPU is idling between the termination of T_ℓ and the start of T_n. Symmetrically, if T_n is executed on GPU, no GPU is idling in this period.

Therefore, when no task of p is being executed, either all the CPU are busy or all the GPU are busy. The CPU (resp. GPU) can be all busy for at most a time of W_c/m (resp. W_g/k). And the tasks of p are executed during a time at most CP.

Hence, we have:

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + CP.$$

We can further refine this inequality. Let P_c (resp. P_g) be the processing time of the tasks of p on CPU (resp. GPU). Then, these processing times can be removed from the total loads in the inequality:

$$\begin{aligned} C_{max} &\leq \frac{W_c - P_c}{m} + \frac{W_g - P_g}{k} + P_c + P_g \\ &\leq \frac{W_c}{m} + \frac{W_g}{k} + \frac{m-1}{m} P_c + \frac{k-1}{k} P_g \\ &\leq \frac{W_c}{m} + \frac{W_g}{k} + \frac{m-1}{m} \left(P_c + P_g\right) \\ &\leq \frac{W_c}{m} + \frac{W_g}{k} + \frac{m-1}{m} CP. \end{aligned}$$

□

Bounding the loads We denote by A_c (resp. A_g) the set of tasks placed on CPU (resp. GPU) by both $\mathscr{S}$ and in the optimal solution. We denote by C_c (resp. C_g) the set of tasks placed on CPU (resp. GPU) by $\mathscr{S}$ but not in the optimal solution. The lowercase denotes the sum of the processing times of these sets.

The optimal makespan OPT is at least equal to the average work on CPU (and on GPU) in the optimal solution. In this solution, the tasks executed on CPU are tasks of the sets A_c and C_g. Tasks of A_c have the same processing time in $\mathscr{S}$ and in the optimal solution, as they are executed on CPU in both cases. Tasks of C_g are completed faster in $\mathscr{S}$ than in the optimal solution. More precisely, the allocation phase ensures that any task T_i of C_g verifies $\overline{p_i} \geq \tau \underline{p_i}$. Therefore, bounding OPT by the average work on CPU gives the following inequality:

$$\text{OPT} \geq \frac{1}{m} \left(a_c + \tau c_g\right). \tag{1}$$

Similarly, bounding OPT by the average work on GPU gives:

$$\text{OPT} \geq \frac{1}{k} \left(a_g + \frac{c_c}{\tau}\right). \tag{2}$$

Using the fact that $\tau^2 \geq m/k \geq 1$, we multiply by τ both sides of Equation (1) to get:

$$\tau \text{OPT} \geq \frac{\tau a_c}{m} + \frac{\tau^2 c_g}{m} \geq \frac{a_c}{m} + \frac{c_g}{k}.$$

We then simplify Equation (2) using that $k\tau \leq m$:

$$\text{OPT} \geq \frac{1}{k} \left(a_g + \frac{c_c}{\tau}\right) \geq \frac{a_g}{k} + \frac{c_c}{m}.$$

Summing these two inequalities, we get:

$$(1+\tau)\,\text{OPT} \geq \frac{a_c + c_c}{m} + \frac{a_g + c_g}{k} \geq \frac{W_c}{m} + \frac{W_g}{k} \tag{3}$$

Bounding the critical path We now bound the length of the critical path produced: every task of this critical path is also scheduled in the optimal schedule, and forms a path. Each task can be accelerated by a factor at most τ in the optimal schedule, so the time dedicated to process this path in the optimal schedule is at least CP/τ. Therefore, we have

$$CP \leq \tau \mathrm{OPT}. \tag{4}$$

Finally, from Lemma 1 we get:

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1 - \frac{1}{m}\right) CP.$$

Equations (3) and (4) lead to:

$$C_{max} \leq (1+\tau)\mathrm{OPT} + \left(\tau - \frac{\tau}{m}\right)\mathrm{OPT}.$$

Using that $\tau^2 \geq m/k$ so $\tau/m \geq 1/k\tau$, we deduce:

$$C_{max} \leq \left(2\tau + 1 - \frac{1}{k\tau}\right)\mathrm{OPT}.$$

□

We now prove that the competitive ratio is almost tight in the following theorem.

Theorem 5. *The competitive ratio of* QA *is at least* $\left(2\tau + 1 - \frac{1}{k}\right)$.

Proof. We let ε be a small processing time and we define $q = (k-1)k$.

Consider a graph composed of $q + mk + 2$ tasks, labeled by T_i for i from 1 to $q + mk + 2$. The first $q + mk + 1$ tasks are all independent. These tasks are composed of four groups:

- The first q tasks have an infinite CPU processing time and a GPU processing time equal to $x = (k-1)/q = 1/k$.
- The next mk tasks have a CPU processing time of $(1+\varepsilon)/k$ and a GPU processing time of $1/\sqrt{mk}$.
- The next task, T_{q+mk+1} has an infinite CPU processing time and a GPU processing time of ε.
- The last task of the graph, T_{q+mk+2} is a successor of T_{q+mk+1}. Its CPU processing time is equal to τ, and its GPU time is equal to $1+\varepsilon$.

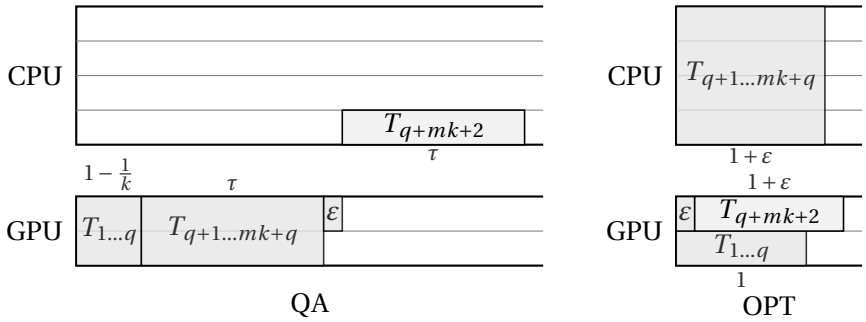


Figure 7: Schedule obtained by QA (left) and the optimal one (right).

We consider the online setting in which the tasks T_i arrive in the order given by i. The ratio of CPU time over GPU time is larger than τ for every task except the last one. Then, QA schedules the first q tasks on k GPUs in time $qx/k = (k-1)/k$. Then, it schedules the next mk tasks on k GPUs in time $m/\sqrt{mk} = \tau$. Task T_{q+mk+1} is then scheduled on GPU in a time ε, after which the last task is scheduled on CPU. The makespan obtained is then equal to:

$$M_{\text{QA}} = 2\tau + \frac{k-1}{k} + \varepsilon.$$

Another possibility consists in scheduling first T_{q+mk+1} then T_{q+mk+2} on a single GPU, which take a time $1+2\varepsilon$. In parallel, tasks T_1 to T_q are scheduled on the remaining $(k-1)$ GPUs, which takes a time $qx/(k-1) = 1$. In parallel, we schedule tasks T_{q+1} to T_{q+mk} on m CPUs, which are then completed at time $1+\varepsilon$. The makespan obtained is then:

$$M = 1 + 2\varepsilon.$$

The schedules obtained are illustrated on Figure 7.

The ratio of the makespan obtained by QA divided by M is then equal to:

$$\frac{M_{\text{QA}}}{M} = \frac{2\tau + \frac{k-1}{k} + \varepsilon}{1+2\varepsilon} \quad \underset{\varepsilon \to 0}{\longrightarrow} \quad 2\tau + \frac{k-1}{k}.$$

□

4.2 A tunable competitive algorithm which performs well in practice

EFT, which stands for Earliest Finish Time, is one of the most intuitive algorithm to solve this problem: it schedules each task on the resource on which it will be completed the soonest. This algorithm has good performance in practice, as the load between resources is maintained balanced. However, on some instances, it can achieve makespans $m/k + 2 - \frac{1}{k}$ times longer than the optimal solution or the one computed by QA, even on independent tasks, see Lemma 2.

Lemma 2. *The competitive ratio of EFT is smaller than* $(m/k + 2 - \frac{1}{k})$*, even on independent tasks.*

Proof. Let ε be arbitrary small. We assume that k divides m and $k > 1$.

We first prove a weaker result, by exposing an instance on which EFT achieves a makespan equal to m/k where the optimal result is $1+\varepsilon$.

Consider $(m+k)m/k$ tasks composed of two types. m tasks of type A have a CPU computing time equal to $1+\varepsilon$ and a GPU computing time equal to 1. The remaining m^2/k tasks, of type B, have a CPU computing time equal to 1 and a GPU computing time equal to ε.

The online instance is decomposed into m/k phases, each starting by k tasks of type A followed by m tasks of type B.

An optimal schedule allocates each task A on a single CPU, and all the tasks B on GPUs. This achieves a makespan equal to $1+\varepsilon$.

EFT allocates the first k tasks A on GPU as they complete faster (1 versus $1+\varepsilon$). Then, all the GPUs are busy until time 1, so EFT allocates the next m tasks of type B on CPU. Therefore, at the end of the first phase, all the processors are busy until time 1. Consequently, after the m/k phases, EFT achieves a makespan equal to m/k. We have then proved the first result.

This instance can now be modified to prove the lemma. Split the last phase into $k-1$ sub-phases, where each sub-phase contains the same tasks, but which computing time are divided by k. EFT schedules this phase in time $1-\frac{1}{k}$, using every processor, achieving a makespan equal to $(m-1)/k$. An optimal schedule uses $k-1$ CPUs to schedule the A tasks in time 1, and schedule the B tasks on GPUs. Now, add a new phase at the end of the instance composed of one task of type A followed by k tasks of type C, which have an infinite CPU computing time and a GPU computing time equal to 1. The schedules obtained are represented in Figure 8. The last A task is noted A' to differentiate it from the previous A tasks.

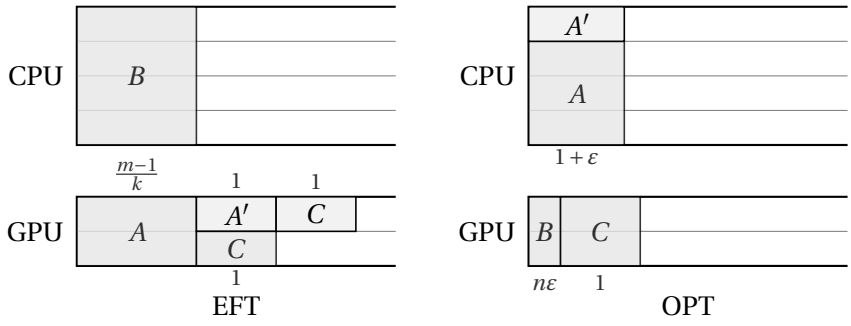


Figure 8: Schedule obtained by EFT (left) and an optimal one (right).

The optimal schedule executes the task A on the last idling CPU, and each task C on a GPU. The makespan obtained is then at most $1+n\varepsilon$, where n is the number of tasks in the graph.

EFT schedules the $k+1$ tasks of this phase on GPU. Its makespan is then increased by 2, to reach a value of $m/k+2-\frac{1}{k}$, hence the lemma. □

We propose a new tunable algorithm, named MIXEFT that benefits both from the performance of EFT on most instances, and from the robustness of QA on the hardest graphs. The idea is to improve EFT by switching to a guaranteed algorithm if EFT does not perform well enough. The algorithm is composed of two phases. In the first phase, it is equal to EFT except that it also simulates the schedule that QA would have produced on the same instance. If the makespan obtained by EFT is more than λ times larger than the makespan obtained by the simulated QA (for a fixed positive parameter λ) we switch to the second phase, and MIXEFT from this point behaves as QA. A small λ leads to a smaller competitive ratio, but may degrade the performance of MIXEFT in practice. We propose to use a value of λ between 1 and 2. The pseudocode is provided in Algorithm 1.

Algorithm 1: MIXEFT (λ)

```
1  P_QA ← simulated platform
2  P_EFT ← simulated platform
3  StayEFT ← yes
4  while there is a new task T_i do
5      if StayEFT then
6          On P_EFT, Schedule T_i as soon as possible on the resource on
             which it completes the earliest
7          On P_QA, schedule T_i as soon as possible on CPU if p̄_i/p_i ≤ τ and
             on GPU otherwise
8          if makespan in P_EFT is λ times larger than the makespan in P_QA
             then
9              StayEFT ← no
10     if StayEFT then
11         Schedule T_i as soon as possible on the resource on which it
             completes the earliest
12     else
13         Schedule T_i as soon as possible on CPU if p̄_i/p_i ≤ τ and on GPU
             otherwise
```

The competitive ratio of this algorithm is in $O(\lambda\tau)$. Indeed, if OPT represents the length of the optimal schedule, QA solves the whole graph in less than $(2\tau+1)$OPT. Therefore, the time to complete the first phase is less than λ times this quantity. For the second phase, it is less than this quantity. The whole graph is then completed in less than $(\lambda+1)(2\tau+1)$OPT.

We however conjecture that the competitive ratio of MIXEFT is similar to $\max(\lambda, 2\tau+1)$. This statement is motivated by two ideas. It seems unlikely

that EFT performs worse than QA on an instance in which QA is far from the optimal. So, when the switch occurs, we expect the makespan to be at most $\max(\lambda, \tau)\text{OPT}$. Secondly, when the switch occurs, it is likely that many resources are busy in the optimal solution. Therefore, we expect the makespan of the optimal solution to increase between the switch and the end of the graph. The competitive ratio is then smaller than the addition of the competitive ratio of both phases.

5 The allocation is more difficult than the schedule

In this section, we focus on the allocation and scheduling phases separately. We assume that one phase is solved by an oracle (i.e., the allocation is fixed or the schedule can be computed offline when the allocation is decided), and we study the problem of minimizing the makespan by computing the focused phase in an online way. The results are summarized in Table 2. In particular, we show that when the allocation is fixed, any online list scheduling algorithm achieves a competitive ratio of $3 - \frac{1}{m}$, which matches the lower bound, and we show that the performance of QA cannot be improved significantly even with an offline scheduling algorithm, as no algorithm can decide the allocation online while guaranteeing a competitive ratio smaller than $\tau = \sqrt{m/k}$.

Online phase	Lower bound	Proof	Upper bound	Proof
Schedule	$3 - \frac{1}{m}$	Lem. 3	$3 - \frac{1}{m}$	Th. 6
Allocation	τ	Th. 7	$< 2\tau + 1$	Th. 4

Table 2: Summary of the results obtained.

5.1 The allocation of each task is fixed

Theorem 6. *If the allocation of each task is fixed, any online list scheduling algorithm is* $\left(3 - \frac{1}{m}\right)$*-competitive.*

Proof. Consider a graph where each task has a fixed alllocation, an online instance of this graph, and the schedule $\mathscr{S}$ computed by any online list scheduling algorithm, of makespan C_{max}. Let W_c (resp. W_g) be the total load on the CPUs (resp. GPUs). Let cp be a critical path of $\mathscr{S}$, and CP be the sum of the processing times of the tasks of cp in $\mathscr{S}$.

The result of Lemma 1 stated in the proof of Theorem 4 holds, therefore we have:

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1 - \frac{1}{m}\right) CP.$$

Let OPT be the optimal makespan given the fixed allocation. The m CPUs have to execute tasks whose execution time sum to W_c, so $\text{OPT} \geq W_c/m$. Simi-

larly, OPT $\geq W_g/k$, and as CP is the length of the critical path, we have OPT $\geq CP$. Therefore, we conclude that:

$$C_{max} \leq \left(3 - \frac{1}{m}\right) \text{OPT}.$$

□

We now show that this upper bound is tight.

Lemma 3. *If the allocation of each task is fixed, no online scheduling algorithm has a competitive ratio smaller than* $\left(3 - \frac{1}{m}\right)$.

Proof. We assume $m \geq 2$. Note that the result also holds for $m = 1$, with a simpler example without the second group of tasks built below. Let $\mathscr{A}$ be an online scheduling algorithm. We let n be an integer multiple of $km(m-1)$ and an adversary will build a graph G composed of the $2n+1$ following tasks:

- tasks T_1 to T_n have a GPU computing time equal to k/n and an infinite CPU computing time
- tasks T_{n+1} to T_{2n} have a CPU computing time equal to $(m-1)/n$ and an infinite GPU computing time
- task T_{2n+1} has a CPU computing time equal to 1 and an infinite GPU computing time

In the graph G, there will exist $i \in [1,n]$ and $j \in [n+1,2n]$ such that the dependences of G are from task T_i to tasks $T_{n+\ell}$ for every $\ell > 0$ and from task T_j to task T_{2n+1}.

Every such graph can be scheduled in time $1 + (k+m-1)/n$: schedule each task as soon as possible starting task T_i at time 0, task T_j at time k/n and task T_{2n+1} at time $(k+m-1)/n$, see Figure 9. Tasks $T_{1\ldots n}$ are completed on k GPUs in time $n/k * k/n = 1$, tasks $T_{n+1\ldots 2n}$ are completed on m CPUs in time $(m-1)/n * n/(m-1) = 1$, and task T_{2n+1} in time 1.

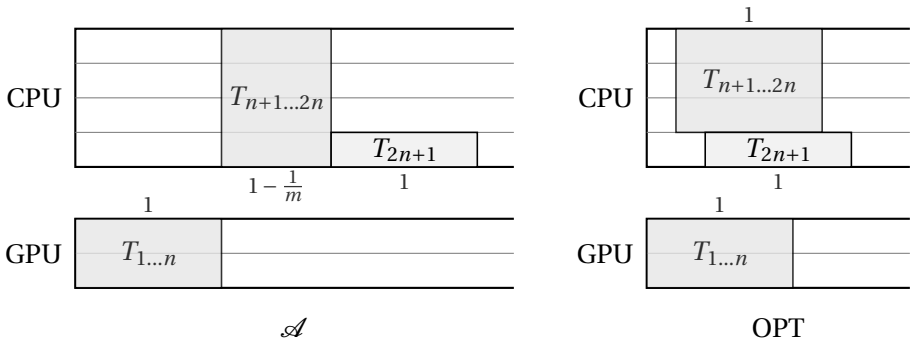


Figure 9: Schedules obtained by $\mathscr{A}$ and OPT.

Now, consider algorithm $\mathscr{A}$. The adversary selects the last task of $T_{1\ldots 2n}$ to be terminated as the predecessor of every task $T_{n+\ell}$ for every $\ell > 0$. Similarly, it selects the last task of $T_{n+1\ldots 2n}$ to be terminated as the predecessor of task T_{2n+1}. The makespan obtained is then at least the time necessary to complete $T_{1\ldots n}$ on k GPUs, plus the time to complete $T_{n+1\ldots 2n}$ on m CPUs, plus the time to complete T_{2n+1} on 1 CPU, see Figure 9:

$$1 + \frac{n}{m}\,\frac{m-1}{n} + 1 = 3 - \frac{1}{m}.$$

Therefore, the competitive ratio of $\mathscr{A}$ is at least:

$$\frac{3-\frac{1}{m}}{1+\frac{1}{n}(k+m-1)} \quad \underset{n\to\infty}{\longrightarrow} \quad 3-\frac{1}{m}.$$

□

5.2 The schedule can be computed offline

We first precisely define this model as it is unusual. At the beginning, the algorithm knows all the available tasks and decides their allocation. Next, the algorithm flags one or several allocated task. Any task whose predecessors have all been flagged is revealed, and the list of its predecessors is known. The algorithm decides the allocation of every revealed tasks, before flagging one or several allocated tasks. The procedure continues until all the tasks have been allocated. The final schedule is then computed offline, but must respect the allocation decided. The objective of this model is therefore to study the complexity of the allocation choice alone. The flagging procedure is used to simulate the termination of tasks.

The algorithm QA fits in this model, as the scheduling phase can be delayed, and has a competitive ratio smaller than $2\tau+1$. We now prove a lower bound for any algorithm deciding the allocation online.

Theorem 7. *No algorithm that decides the allocation online and computes the schedule offline has a competitive ratio smaller than τ.*

Before proving this theorem, we start by a weaker result, which will be reused in the main proof.

Lemma 4. *No algorithm that decides the allocation online and computes the schedule offline has a competitive ratio smaller than $\frac{\tau}{2}$.*

Proof. Consider an algorithm $\mathscr{A}$ with online allocation. We assume that τ is an integer. We will use an adversary proof, by building a graph composed of two types of task denoted T_i and ε_i. For the T_i task, the CPU processing time equals τ and the GPU processing time equals 1. For the ε_i task both CPU and GPU processing times equal ε.

The only tasks without predecessor are tasks ε_1 and T_1. During the procedure, at any time, only one task has undisclosed successors: either ε_i or T_i has two successors ε_{i+1} and T_{i+1}. This adversary chooses which one in function of the allocation of T_i: if T_i is scheduled on CPU, then T_i has successors, if T_i is scheduled on GPU, then ε_i has successors. Intuitively, the tasks ε_i are only used to hide the successors. Note that in our model, the algorithm $\mathscr{A}$ must decide the allocation of both tasks T_i and ε_i before gaining any information on the rest of the graph. So the adversary is able to select the successors of task ε_i according to the allocation of task T_i. See Figure 10 for an example of built graph.

Let C (resp. G) be the set of tasks T_i scheduled on CPU (resp. GPU) by $\mathscr{A}$. The allocation of tasks ε_i is not relevant. We consider the first iteration at which $|C| = \tau$ or $|G| = m$.

We define a schedule $\mathscr{S}$ of the built graph which achieves a makespan at most $2\tau + \varepsilon(m+\tau)$. In order to simplify the reasoning, we always execute tasks ε_i as soon as possible on any available resource. Therefore, describing only the schedule of tasks T_i suffices. Schedule the tasks of C sequentially on a single GPU, which takes at most τ units of time, plus the time necessary to compute the relevant ε_i tasks. Then, schedule the tasks of G, each on a single CPU, after the completion of every ε_i task. This takes a time at most τ. No task of G has any successor, so this schedule is valid. We obtain the upper bound on the makespan reached by summing these two phases and adding the time to compute every ε_i task, see Figure 10.

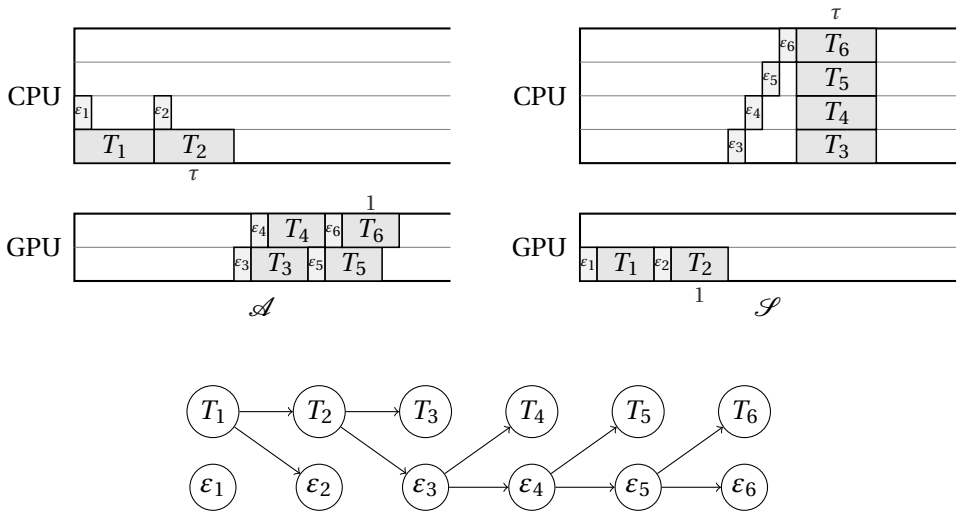


Figure 10: Example of schedule obtained by $\mathscr{A}$ and $\mathscr{S}$ when $\mathscr{A}$ allocates tasks T_1 and T_2 to CPU, and the four following tasks to GPU. Below: graph built by the adversary.

We now show that the schedule of $\mathscr{A}$ has a makespan not smaller than m/k. First, suppose that $|C| = \tau$. Then, the tasks of C must be executed sequentially

as they form a path in the graph, and completing one task takes a time τ, so the makespan obtained is at least $\tau^2 = m/k$. Otherwise, we have $|G| = m$. Each task of G is completed in one unit of time, so scheduling the whole set G on k GPUs takes at least m/k units of time. Therefore, the competitive ratio of $\mathscr{A}$ is at least:

$$\frac{m/k}{2\tau + \varepsilon(m+\tau)} \quad \xrightarrow[\varepsilon \to 0]{} \quad \frac{\tau}{2}.$$

□

Proof of Theorem 7. This proof is similar to the one of Lemma 4, except that we further extend the analysis, and we do not focus on the ε tasks for the schedule, as their completion time can be ignored by choosing a small value for ε. We fix an integer n as large as we want. A large n will lead to a large graph and a competitive ratio closer to τ.

As previously, we consider an algorithm $\mathscr{A}$ with online allocation, and we use an adversary proof with the exact same strategy.

Let C (resp. G) be the set of tasks T_i scheduled on CPU (resp. GPU) by $\mathscr{A}$. The end of the procedure will be specified later, but will respect $|C| \leq n\tau$ and $|G| \leq mn$.

We define a schedule $\mathscr{S}$ of the built graph. In a first step, we assume that $\varepsilon = 0$ so we do not focus on the allocation of the ε_i tasks. We will come back to the complete graph including ε_i tasks when computing the total makespan of $\mathscr{S}$. We define a *bucket* as a set of processors, a starting time and a duration time. We use buckets to reserve some processors for an amount of time, and schedule a set of tasks in a given bucket. We consider $n+1$ buckets, as illustrated in Figure 11. Each of the B_i buckets (for $i = 1 \dots n$) concerns all m CPUs, starts at time $i\tau$ and has duration τ. All tasks of G will be scheduled by $\mathscr{S}$ into some bucket B_i. Note that m tasks of G fit into each bucket. The last bucket, B concerns one GPU, starts at time 0 and lasts a time $n\tau$. Tasks of C will be scheduled by $\mathscr{S}$ into bucket B. Note that all tasks of C fit into this bucket.

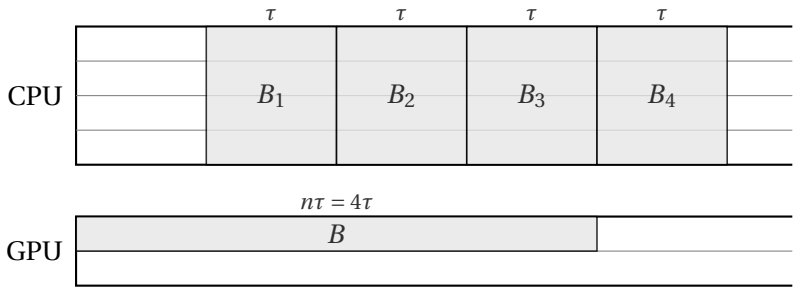


Figure 11: Buckets used by $\mathscr{S}$ with $n = 4$.

In the schedule $\mathscr{S}$, the first tasks of G are allocated to bucket B_1, the following ones to bucket B_2 and so on. We pass from bucket B_i to bucket B_{i+1} when

either bucket B_i is full (contains m tasks) or bucket B contains $i\tau$ tasks. The procedure stops when either bucket B is full (contains $n\tau$ tasks) or the last bucket B_n is full. Note that one of these events will eventually happen: by adding new tasks, if B never gets full, all tasks are added to G and fill B_n. See Figure 12 for an illustration.

We first show that $\mathscr{S}$ is a valid schedule. By definition, no bucket is overloaded. Concerning the precedence relations, only tasks of C have successors. So the tasks of C can be scheduled into bucket B without idle time (still assuming $\varepsilon = 0$). When a given task of C is scheduled, all the subsequent tasks of G are scheduled after its termination, as we stop allocating tasks in bucket B_i, which starts at time $i\tau$, as soon as $i\tau$ tasks of C have been scheduled. See Figure 12 for an illustration. Here, $\tau = 2$, so as after T_4, the second task of C is executed, the next tasks of G are scheduled into the second bucket. Therefore, no precedence constraints are violated. Note that we could delay the moment where we move to the next bucket until the next task of C is executed, here T_6. Indeed, task T_5 could be executed in the first bucket.

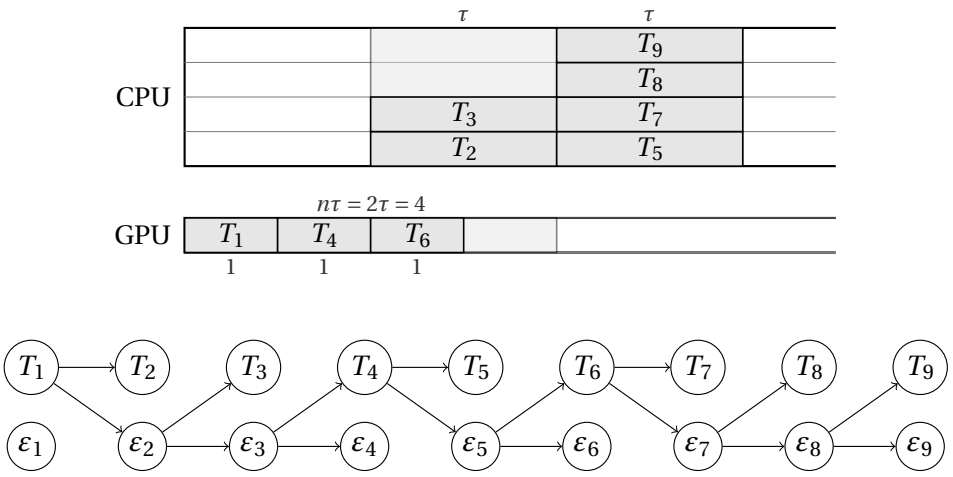


Figure 12: Example of schedule obtained by $\mathscr{S}$ with four CPUs, one GPU, $n = 2$, and when $\mathscr{A}$ places T_1, T_4, and T_6 on CPU. Below: graph built by the adversary.

Therefore, $\mathscr{S}$ completes the graph in a makespan of at most $(n+1)\tau$. Recall that we have omitted at most $n(m+\tau)$ tasks ε_i in this computation. We then add their total computation time to the actual makespan obtained: $(n+1)\tau + n(m+\tau)\varepsilon$.

Now, we consider algorithm $\mathscr{A}$, and we show that the makespan obtained is at least $n\tau^2$. We first treat corner cases. If $|C| = n\tau$ at the end of the procedure, then tasks of C are scheduled sequentially on a CPU and the makespan obtained by $\mathscr{A}$ is at least $\tau|C| = n\tau^2$. If $|G| = mn$ at the end of the schedule, then tasks of G are scheduled on k GPUs and the makespan obtained by $\mathscr{A}$ is at least $|G|/k = n\tau^2$.

Then, we suppose that $|C| < n\tau$, and $|G| < mn$. Thus, there exists a bucket B_i which is not full, and the last bucket B_n is full. Let j be the index of the last bucket B_j which is not full, so $1 \leq j \leq n-1$. Therefore, B contains at least $j\tau$ tasks, and $j\tau$ tasks of B have been scheduled before bucket B_{j+1} has been allocated any task. As all the following buckets B_i for $i > j$ are full, exactly $m(n-j)$ tasks of G are scheduled in these buckets. Therefore, the graph obtained contains a path of $j\tau$ tasks of C followed by $m(n-j)$ tasks of G. In Figure 12, recall that $n = 2$ and $\tau = 2$. We have $j = 1$, as the first bucket B_1 is not full, therefore we have a path of $j\tau = 2$ tasks of C (namely T_1, T_4), followed by $m(n-j) = 4$ tasks of G (namely T_5, T_7, T_8, T_9). Algorithm $\mathscr{A}$ completes these tasks of C in time at least $j\tau^2$, and these tasks of G in time at least $m(n-j)/k = n\tau^2 - j\tau^2$. Finally, the makespan of $\mathscr{A}$ is at least $n\tau^2$.

The competitive ratio of $\mathscr{A}$ is then at least

$$\frac{n\tau^2}{(n+1)\tau + n(m+\tau)\varepsilon} = \frac{\tau}{1 + \frac{1}{n} + \varepsilon\left(\frac{m}{\tau}+1\right)} \quad \underset{\varepsilon \to 0}{\overrightarrow{n\to\infty}} \quad \tau.$$

□

6 Extension to multiple types of processors

We generalize our study to $Q \geq 2$ types of processors. Note that in the offline setting, Amaris et al. [2] provide a $Q(Q+1)$-approximation. We denote by m_q be the number of processors of type q, and we assume that they are ordered such that $m_q \geq m_{q+1}$. For a task T_i, $p_{i,q}$ denotes its computing time on processor type q.

Our first result extends the lower bounds of Section 3 for Q processor types.

Theorem 8. *Any online algorithm for Q processor types has a competitive ratio smaller than $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q}$.*

Proof. Let $\mathscr{P}$ be the targeted platform composed of Q types of processors. Consider an alternative platform $\mathscr{P}'$ composed of 2 types of processors, with $m' = \sum_{q=1}^{Q-1} m_q$ CPUs and $k' = m_Q$ GPUs.

Any instance G' on $\mathscr{P}'$ can be simulated by an instance G on $\mathscr{P}$: for any task T_i of the graph, let $p_{i,Q}$ be equal to the GPU processing time of T_i on $\mathscr{P}'$ and let $p_{i,q}$, for $q = 1, \ldots, Q-1$ be equal to its CPU processing time. Therefore, G can be scheduled in time M on $\mathscr{P}$ if and only if G' can be scheduled in time M on $\mathscr{P}'$: the processor types 1 to $Q-1$ are equivalent in $\mathscr{P}'$ and can be mapped to the CPUs of $\mathscr{P}$.

Suppose by contradiction that an online algorithm has a competitive ratio smaller than $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q} = \sqrt{m'/k'}$ on $\mathscr{P}$. Its competitive ratio on $\mathscr{P}'$ is then smaller than $\sqrt{m'/k'}$, which violates Theorem 1. □

The same generalization can be done for every lower bound presented in Table 1.

We also adapt our QA algorithm for this setting, by changing its allocation phase:

- Allocate T_i to processor type q such that $p_{i,q} \big/ \sqrt{m_q}$ is minimal.

Note that with $Q = 2$, this algorithm is equivalent to the original QA. The following theorem (proved below) generalizes the competitive ratio.

Theorem 9. *On Q types of processors, the approximation ratio of* QA *is at most* $\frac{1}{\sqrt{m_Q}}\left(\sqrt{m_1} + \sum_{q=1}^{Q}\sqrt{m_q}\right)$.

In comparison, there is an instance similar to the one of Lemma 2 on which EFT achieves a ratio larger than $\sum_{q=1}^{Q} m_q/m_Q$. Indeed, by setting identical computing times to processor types 1 to $Q-1$, EFT behaves as if there were $\sum_{q=1}^{Q-1} m_q$ CPUs and m_Q GPUs, which leads to this result.

The lower bounds proved in Section 3 are extended by replacing τ by $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q}$, using the same technique.

Proof of Theorem 9. This proof is similar to the one of Theorem 4.

We consider a graph, an online instance of this graph, and the schedule $\mathscr{S}$ computed by QA, of makespan C_{max}. We consider also an optimal solution, to which we will refer as *the* optimal solution, of makespan OPT. Let W_q be the total load on the processors of type q for each q. Let *cp* be a critical path of $\mathscr{S}$, and *CP* be the sum of the processing times of the tasks of *cp* in $\mathscr{S}$.

First, we prove that:

$$C_{max} \leq \sum_{q=1}^{Q} \frac{W_q}{m_q} + CP. \tag{5}$$

As in Theorem 4, consider a path of tasks p form the graph whose execution starts the soonest and terminates exactly at time C_{max} in $\mathscr{S}$. When no task of p is being executed, one type of processor is necessarily busy because of the scheduling strategy, which always schedules an available task if one processor of every type is idling. The total amount of time during which at least one type of processor has no idle resource is at most $\sum_{q=1}^{Q} \frac{W_q}{m_q}$, hence the result.

We now bound *CP*. Consider a task T_i that is executed on processor type ℓ in QA and q in the optimal solution. We have, by definition of QA, m_1 and m_Q:

$$p_{i,\ell} \leq \sqrt{\frac{m_\ell}{m_q}}\, p_{i,q} \leq \sqrt{\frac{m_1}{m_Q}}\, p_{i,q}. \tag{6}$$

Summing over the tasks of *cp*, we obtain:

$$CP \leq \sqrt{\frac{m_1}{m_Q}}\mathrm{OPT}. \tag{7}$$

We consider the workload W_q^* on processor type q in the optimal solution, which is not larger than $m_q\text{OPT}$. For any processor type ℓ, let C_q^ℓ be the sum of the computing times on processors of type ℓ of tasks allocated to processor type ℓ in QA and to processor type q in the optimal solution. We can lower bound W_q^* in the optimal solution by the quantities C_ℓ^q, using the first inequality of Equation (6):

$$\text{OPT} \geq \frac{W_q^*}{m_q} \geq \frac{1}{m_q} \sum_{\ell=1}^{Q} \sqrt{\frac{m_q}{m_\ell}} C_q^\ell$$
$$\sqrt{\frac{m_q}{m_Q}}\text{OPT} \geq \sum_{\ell=1}^{Q} \frac{C_q^\ell}{\sqrt{m_Q m_\ell}} \geq \sum_{\ell=1}^{Q} \frac{C_q^\ell}{m_\ell}.$$

Now, summing over all processor types q, we get

$$\begin{aligned}\frac{1}{\sqrt{m_Q}}\left(\sum_{q=1}^{Q}\sqrt{m_q}\right)\text{OPT} &\geq \sum_{q=1}^{Q}\sum_{\ell=1}^{Q}\frac{C_q^\ell}{m_\ell} \\ &\geq \sum_{\ell=1}^{Q}\frac{1}{m_\ell}\sum_{q=1}^{Q} C_q^\ell \\ &\geq \sum_{\ell=1}^{Q}\frac{W_\ell}{m_\ell}. \end{aligned} \tag{8}$$

Finally, combining Equations (5), (7) and (8), we get the result:

$$C_{max} \leq \frac{1}{\sqrt{m_Q}}\left(\sqrt{m_1} + \sum_{q=1}^{Q}\sqrt{m_q}\right)\text{OPT}.$$

□

7 Simulations

We now provide simulations to illustrate the performance of both competitive algorithms and simple heuristic strategies on various task graphs.

7.1 Baseline heuristics

In addition to the four online algorithms discussed above (ER-LS from [3], QA, EFT, and MixEFT, implemented with $\lambda = 2$ unless otherwise specified), we consider two simple strategies that follow the same scheme as QA, with a different allocation criteria: QUICKEST allocates each task to the resource type on which its computing time is smaller; RATIO allocates a task on GPUs if and only if its GPU computing time is at least m/k times smaller than its CPU computing time. Intuitively, QUICKEST should perform well on graphs on which the critical path

is preponderant. On the opposite, RATIO should perform well on graphs with a high parallelism throughout the execution.

We also used the offline HEFT algorithm [22], which is known to perform well in practice, as a baseline to compare all online strategies. Note that the priority of each task in HEFT is computed using the average of the costs on all the cores (average of the CPU and GPU costs weighted by the number of each type of resources). Moreover, backfilling is performed following HEFT insertion policy.

7.2 Experimental setup

We used three types of instances: realistic DAGs corresponding to the Cholesky factorization, random DAGs used in the literature, and ad hoc instances designed to be difficult for this problem and specifically for QA.

Cholesky factorization is a linear algebra application whose parallel implementation usually uses a blocked algorithm on a tiled matrix for performance reasons. We consider matrix sizes ranging from 2×2 tiles to 15×15 tiles, which leads to DAGs with 4 to 680 tasks. Tasks correspond to four linear algebra kernels: GEMM, SYRK, TRSM, and POTRF. Their respective processing times on a CPU are set to 170ms, 95ms, 88ms, and 33ms, and on a GPU to 5.95ms, 3.65ms, 8.11ms, and 15.6ms, which corresponds to measures [1, 7] made using the Chameleon software [10].

The random instances come from the STG set [20], which is often used in the literature to compare the performance of scheduling strategies. The set contains instances with 50 to 5000 nodes. We report here the simulations made with 180 graphs of 300 nodes each. In these instances, 45 graphs are generated by each random DAG generator (`layrpred`, `layrprob`, `samepred` and `sameprob`). Both `layrpred` and `layrprob` generators lead to graphs with nodes structured by layers, whereas `samepred` and `sameprob` lead to more intricate graphs. In contrast to their counterpart with the suffix `-prob`, generators with the suffix `-pred` specify the average number of predecessors for each task. We consider that the cost generated by the STG random generator is the processing time of the corresponding task on a GPU. Based on the previous measures for linear algebra kernels, we assume that the average speedup between CPU and GPU is around 15 with a large variance. Thus, to obtain the processing time of a task on CPU, we multiply its cost on GPU by a random value with expected value 15 and standard deviation 15. For that, we use a gamma distribution because it has been advocated for modeling job runtimes [13], it is positive and it is possible to specify its expected value and standard deviation by adjusting its parameters.

Finally, specific random instances have been designed to test the limitations of QA. These ad hoc instances consist of a chain of tasks together with a set of independent tasks, such that all cores are expected to finish simultaneously if a GPU is dedicated to the chain and all independent tasks are load-balanced on the other cores. The expected processing time of a task on a GPU is 1 (with

a standard deviation of 0.1) and the expected processing time on a CPU varies from $(m/k)^{-1/4}$ to $(m/k)^{5/4}$ (with a standard deviation equal to 10% of this expected value). For a given expected CPU cost μ, the number of tasks in the chain is $\lceil \frac{n}{m/\mu+k} \rceil$, where $n = 300$ is the total number of tasks. Therefore, the larger μ, the longer the chain.

The code and scripts used for the simulations and the data analysis are available online at `https://doi.org/10.6084/m9.figshare.5919241`.

7.3 Results

Figures 13 to 16 depict the performance of the six online scheduling algorithms for $m = 20$ CPUs and $k = 2$ GPUs because it best highlights the difference between the online strategies. Except when varying its parameter λ (Figure 16), MixEFT performs exactly as EFT (and is thus omitted for better readability).

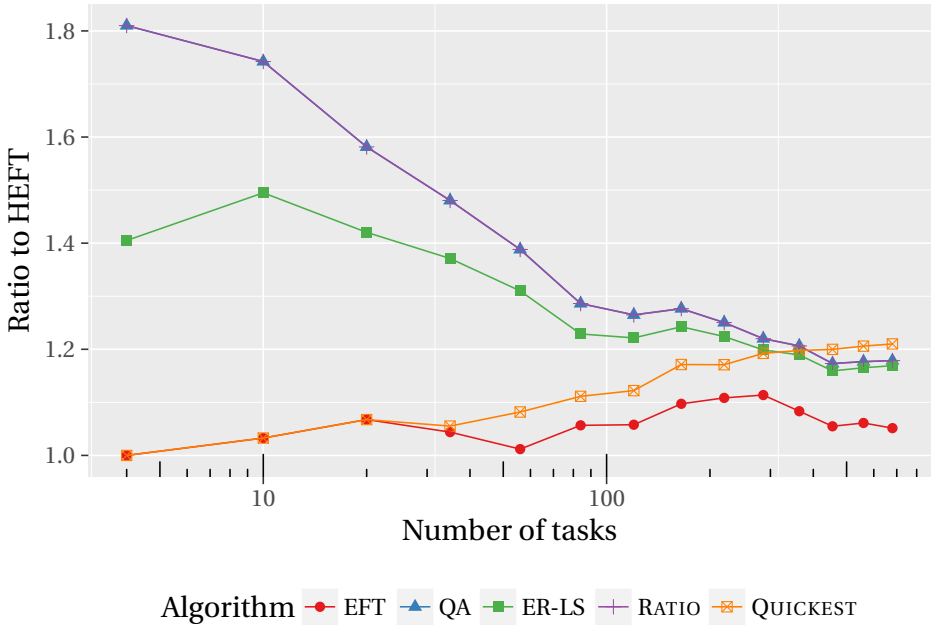


Figure 13: Ratios of the makespan over HEFT for EFT, QA, ER-LS, Ratio, and Quickest with $m = 20$ CPUs and $k = 2$ GPUs on Cholesky instances. MixEFT is not shown because it performs exactly as EFT.

On Cholesky DAGs (Figure 13), EFT (and thus MixEFT) is always the best strategy. The only difference between QA and ER-LS concerns the first tasks (as we removed Step 1a in QA), which explains why their behavior is similar for large graphs. QA, ER-LS, and Ratio all put POTRF tasks on a CPU, which leads to performance loss when the graph is small because its parallelism is limited and the GPUs are often idle. However, it is acceptable for larger graphs in which

many tasks may be executed in parallel on the GPUs. On the contrary, QUICKEST puts all tasks on the GPUs. This is efficient for small graphs with low parallelism but it becomes worse than RATIO for large graphs.

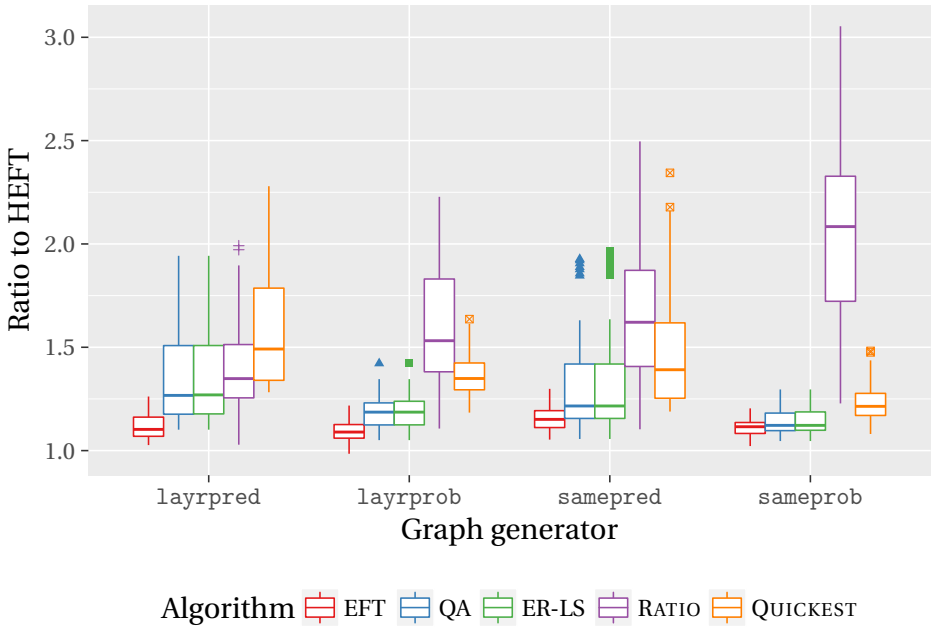


Figure 14: Ratios of the makespan over HEFT for EFT, QA, ER-LS, RATIO, and QUICKEST with $m = 20$ CPUs and $k = 2$ GPUs on random instances with $n = 300$ tasks from the STG data set. MIXEFT is not shown because it performs exactly as EFT.

Figure 14 shows that `layrpred` graphs from the STG data set yield the largest difference between QA/ER-LS and the best algorithms (HEFT and EFT). Additionally, RATIO is often better than QUICKEST. This suggests that using additional CPUs increases the efficiency and that `layrpred` graphs have some parallelism. `sameprob` graphs leads to opposite conclusions because QUICKEST performs well. Contrarily to `layrpred` graphs in which each layer becomes ready step by step, allowing the CPUs to execute some of the tasks without slowing down the GPUs, `sameprob` graphs have more intricate dependences that provide limited parallelism.

Figure 15 first shows that EFT (and MIXEFT) is almost always the best online heuristic on the ad hoc graphs. For extreme values of the expected CPU processing time μ (significantly smaller than 1 or larger than m/k), all four other heuristics are equivalent and perform well. Otherwise, when μ is slightly larger than 1, the instance contains many independent tasks and QUICKEST is almost m/k worst than HEFT because scheduling the independent tasks on GPUs is not efficient. Symmetrically, when μ is slightly smaller than m/k, the instance contains

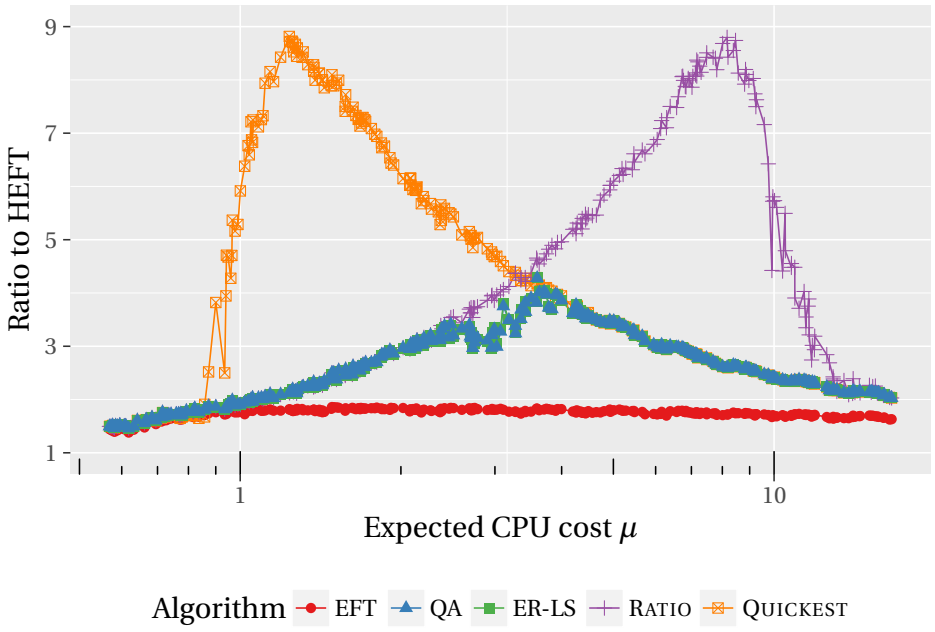


Figure 15: Ratios of the makespan over HEFT for EFT, QA, ER-LS, RATIO, and QUICKEST with $m = 20$ CPUs and $k = 2$ GPUs on 300 ad hoc instances with $n = 300$ tasks. MIXEFT is not shown because it performs exactly as EFT.

a large critical path and RATIO shows poor performance, because it schedules the critical path on CPUs. QA and ER-LS take the best of these two strategies, and have a worst performance $\sqrt{m/k} \approx 3$ times larger than HEFT, when μ is close to $\sqrt{m/k}$.

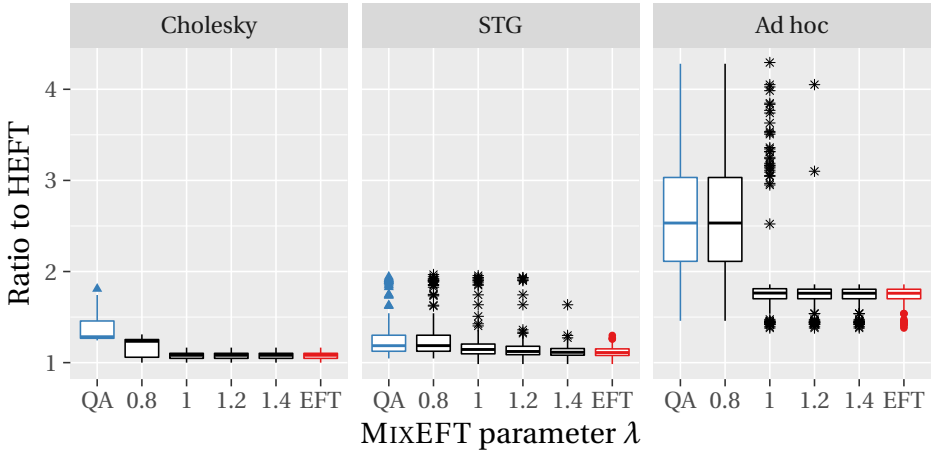


Figure 16: Ratios of the makespan over HEFT for QA, MIXEFT, and EFT with $m = 20$ CPUs and $k = 2$ GPUs on 14 Cholesky, 180 STG, and 300 ad hoc instances. ER-LS, RATIO, and QUICKEST are discarded.

Figure 16 shows that MIXEFT behaves like QA when its parameter λ is smaller than 1, and rapidly changes to mimic EFT when the parameter increases and exceeds 1. This transition occurs for a lower λ for Cholesky instances than for STG and ad hoc ones.

Figure 17 shows the performance for various platform sizes for the Cholesky dataset. EFT is always the best online heuristic and its ratio to HEFT is never more than 1.2 (i.e., 20% worse than HEFT). This also applies to MIXEFT. Depending on the number of CPUs and GPUs, the other algorithms (QA, ER-LS, RATIO, and QUICKEST) follow one of the following three strategies: 1) all tasks on CPUs – this is the case for RATIO when $m/k = 20$; 2) POTRF tasks (the least accelerated tasks) on CPUs and other tasks on GPUs – this is the case for QA and ER-LS when $m/k \geq 5$, and RATIO when $3 \leq m/k \leq 10$; 3) all tasks on GPUs – all the other cases. This first strategy is the worst one except when there are many tasks and CPUs, and a single GPU. In this case, it outperforms the third strategy because the instances present a large parallelism for which CPUs can be exploited. The second strategy is often inefficient for small instances because POTRF tasks are on the critical path and benefit from being accelerated on the GPUs. Finally, the last strategy significantly deviates from EFT only for low k and large number of tasks, which suggests that it is advantageous to exploit CPUs for large graphs when there are few GPUs.

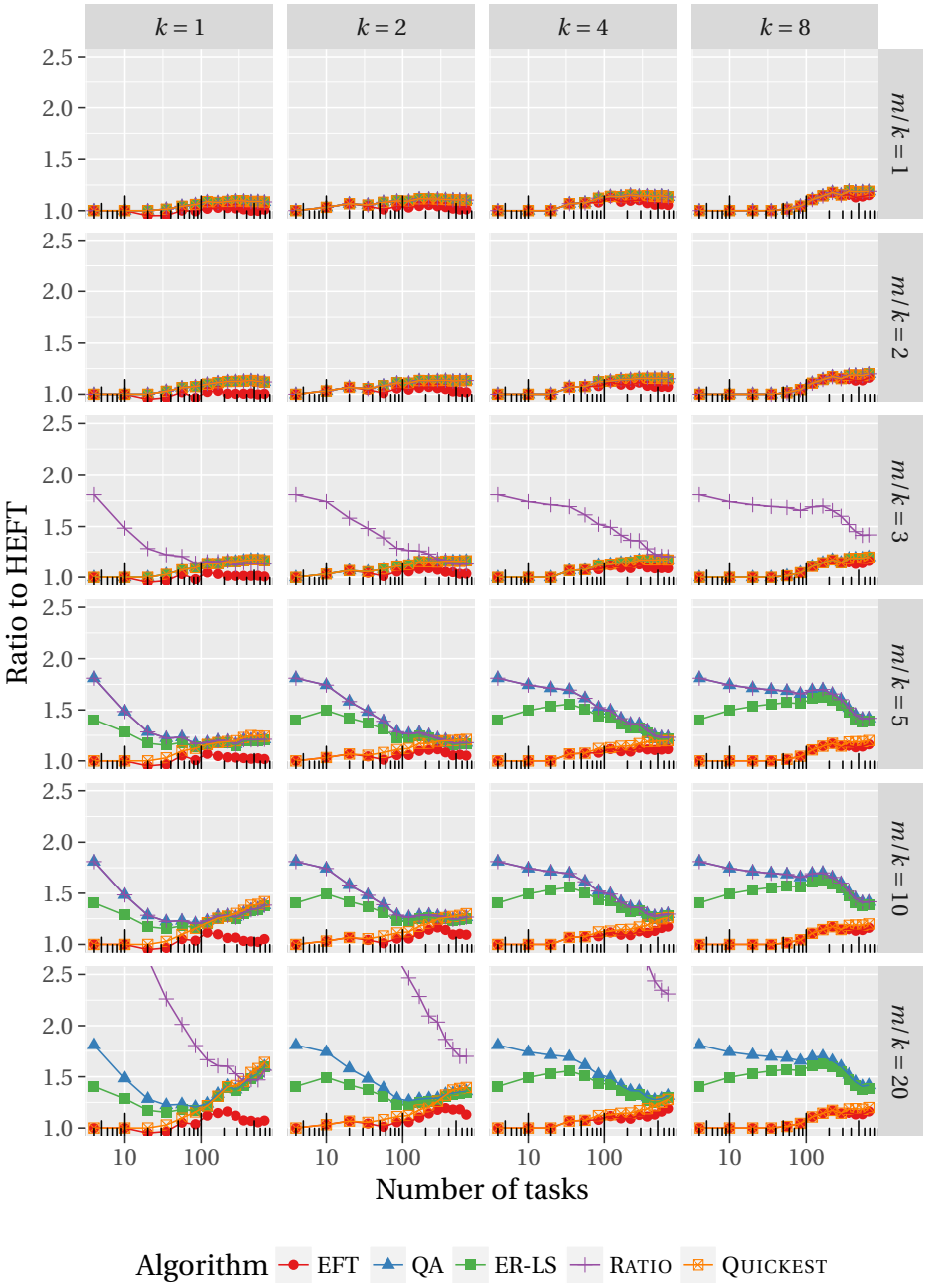


Figure 17: Ratios of the makespan over HEFT for EFT, QA, ER-LS, RATIO, and QUICKEST on Cholesky instances. MIXEFT is not shown because it performs exactly as EFT. In the bottom-right plot, RATIO does not appear because its ratio is too large.

Note that in all studied instances, EFT was never far from HEFT and that there is no practical gain of using MIXEFT rather than EFT. The main advantage of MIXEFT lies in its competitive ratio whereas EFT can lead to very large makespans on specific instances.

8 Conclusion

In this paper, we have focused on the problem of scheduling task graphs on hybrid platforms made of two types of processors, such as CPUs and GPUs. We have studied the online case, when only the tasks whose predecessors are all completed are known to the scheduler, and the graph is thus gradually discovered. We proved that no scheduling algorithm can have a competitive ratio smaller than $\sqrt{m/k}$, and studied how this ratio varies when more knowledge on the graph is given to the scheduler and/or tasks may be migrated between processors. We have proposed a $(2\sqrt{m/k}+1)$-competitive algorithm as well as a mixed strategy, which is both $\Theta(\sqrt{m/k})$-competitive and performs as well as the best heuristics in practice. This is demonstrated through an extensive set of simulations. We have also extended the lower bounds and the competitive algorithms to the case with more than two types of processors. Our future work includes taking into account communication times when moving data from/to the GPUs, and coping with inaccurate processing time estimates.

References

[1] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. Are Static Schedules so Bad? A Case Study on Cholesky Factorization. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 1021–1030. IEEE, 2016.

[2] Marcos Amaris, Giorgio Lucarelli, Clément Mommessin, and Denis Trystram. Generic algorithms for scheduling applications on heterogeneous multi-core platforms. Technical report, CoRR, 2017.

[3] Marcos Amaris, Giorgio Lucarelli, Clément Mommessin, and Denis Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *Euro-Par 2017: Parallel Processing*, pages 220–231, 2017.

[4] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 291–298, Dec 2010.

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heteroge-

neous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[6] O. Beaumont, L. Eyraud-Dubois, and S. Kumar. Approximation proofs of a fast and efficient list scheduling algorithm for task-based runtime systems on multicores and GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 768–777, 2017.

[7] Olivier Beaumont, Terry Cojean, Lionel Eyraud-Dubois, Abdou Guermouche, and Suraj Kumar. Scheduling of linear algebra kernels on multiple heterogeneous resources. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2016.

[8] Raphaël Bleuse, Thierry Gautier, João VF Lima, Grégory Mounié, and Denis Trystram. Scheduling data flow program in XKaapi: A new affinity based algorithm for heterogeneous architectures. In *Euro-Par 2014: Parallel Processing*, pages 560–571, 2014.

[9] Louis-Claude Canon, Loris Marchal, and Frédéric Vivien. Low-cost approximation algorithms for scheduling independent tasks on hybrid platforms. In *Euro-Par 2017: Parallel Processing*, pages 232–244, 2017.

[10] Chameleon, a dense linear algebra software for heterogeneous architectures. `https://project.inria.fr/chameleon`.

[11] Lin Chen, Deshi Ye, and Guochuan Zhang. Online scheduling of mixed CPU-GPU jobs. *International Journal of Foundations of Computer Science*, 25(06):745–761, 2014.

[12] Maciej Drozdowski. Scheduling parallel tasks – algorithms and complexity. In Joseph Leung, editor, *Handbook of Scheduling*. Chapman and Hall/CRC, 2004.

[13] D Feitelson. Workload modeling for computer systems performance evaluation. *Book Draft, Version 1.0.1*, pages 1–601, 2014.

[14] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[15] Csanad Imreh. Scheduling problems on two sets of identical machines. *Computing*, 70(4):277–294, 2003.

[16] S. Kedad-Sidhoum, F. Monna, and D. Trystram. Scheduling tasks with precedence constraints on hybrid multi-core machines. In *IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 27–33, 2015.

[17] Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. Scheduling independent tasks on multi-cores with GPU accelerators. In *Euro-Par 2013: Parallel Processing Workshops*, pages 228–237, 2014.

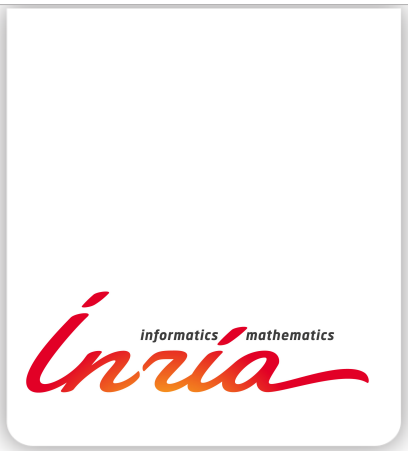
[18] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[19] Florentino Sainz, Sergi Mateo, Vicenç Beltran, José Luis Bosque, Xavier Martorell, and Eduard Ayguadé. Leveraging OmpSs to exploit hardware accelerators. In *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 112–119, 2014.

[20] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394, 2002.

[21] TOP500 Supercomputer Site. `http://www.top500.org`, List of November 2017.

[22] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399